ReadMe

Business Rules:
Business rules classes are stored in separate `package server.businessRules`. Based on Open Closed Principle I use two Interfaces, one for map related rules `IMapRule` and one for player related `IPlayerRule`. These interfaces are later extended by the real specific business rule classes.

Classes for player rules: `PlayerMaySendMapHalf, PlayerMaySendMove, PlayerSentMapHalfOnce.`

Classes for map rules: `HalfMapHasEnoughTerrainTypes, HalfMapHasEnoughTiles, HalfMapHasNoIslands, HalfMapHasNoWaterOnBorders, HalfMapHasOneFort, HalfMapHasValidBordersLength.`

These Rules are later called by logic classes PlayerManager and MapManager to validate the received from client information.

Network Communication
To create communication between client and server the REST interface was implemented. There are 4 implemented endpoints:

- **`http(s)://<domain>:<port>/games` GET**
- **`http(s)://<domain>:<port>/games/<SpielID>/players` POST**
- **`http(s)://<domain>:<port>/games/<SpielID>/halfmaps` POST**
- **`http(s)://<domain>:<port>/games/<SpielID>/states/<SpielerID>` GET**

Each endpoint has one specific task according to the single responsibility principle.

1. First endpoint `newGame` creates a new game using `GameManager` class from logic package, that returns `GameIdentifier` which is later converted to expected `UniqueGameIdentifier` and sent to the client.

2. Second endpoint `registerPlayer` converts received `PlayerInfo` using `ClientConverter` and transfers it to `PlayerManager` class that registers new player and chooses a random Player, who has to send Map first, when both players are registered. The function returns `UniquePlayerIdentifier.`

3. Third endpoint `supplementMap` is receiving half maps and creates a full game map out of them. It calls `playerManager.checkPlayerBusinessRulesForHalfMapPost(player)` to check, if the player may send the map half and if they may, calls `mapManager` class to validate and set the full map. At last the playerManager is called to set the status of player, who must send Move and returns `ERequestState.`

4. Fourth endpoint `sendGameState` calls PlayerManager class to `setFakeEnemyID` and collect all the necessary information about the state of the game and convert it for client, including hiding the enemy position, castle and treasures. The method returns the adapted individually to certain client GameState.

Bonus Points

I used Test Driven Development principle during implementation of four IMap related business rule classes. My implementation had the following structure, from bottom to top:

TDD: Refactor. Refactor HalfMapHasValidBordersLength class for better readability.
TDD: Refactor. Refactor HalfMapHasNoWaterOnBorders class for better readability.
TDD: Refactor. Refactor HalfMapHasNoIslands class for better readability.
TDD: Refactor. Refactor HalfMapHasEnoughTerrainTypes class for better readability.

TDD: Green. Implement HalfMapHasValidBordersLength class.
TDD: Green. Implement HalfMapHasNoWaterOnBorders class.
TDD: Green. Implement HalfMapHasNoIslands class.
TDD: Green. Implement HalfMapHasEnoughTerrainTypes class.

TDD: Red. Implement HalfMapHasValidBordersLengthTest Tests.
TDD: Red. Implement HalfMapHasNoWaterOnBordersTest Tests.
TDD: Red. Implement HalfMapHasNoIslandsTest Tests.
TDD: Red. Implement HalfMapHasEnoughTerrainTypes Tests.

TDD: Think. Add Skeleton for HalfMapHasValidBordersLength rule.
TDD: Think. Add Skeleton for HalfMapHasNoWaterOnBorders rule.
TDD: Think. Add Skeleton for HalfMapHasNoIslands rule.
TDD: Think. Add Skeleton for HalfMapHasEnoughTerrainTypes rule.